# Queston

Given the root of a binary tree, return *the postorder traversal of its nodes' values*.

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [3,2,1]

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Example 4:**



**Input:** root = [1,2]

**Output:** [2,1]

**Example 5:**



**Input:** root = [1,null,2]

**Output:** [2,1]

**Constraints:**

* The number of the nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

**Follow up:**

Recursive solution is trivial, could you do it iteratively?

# Solution

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

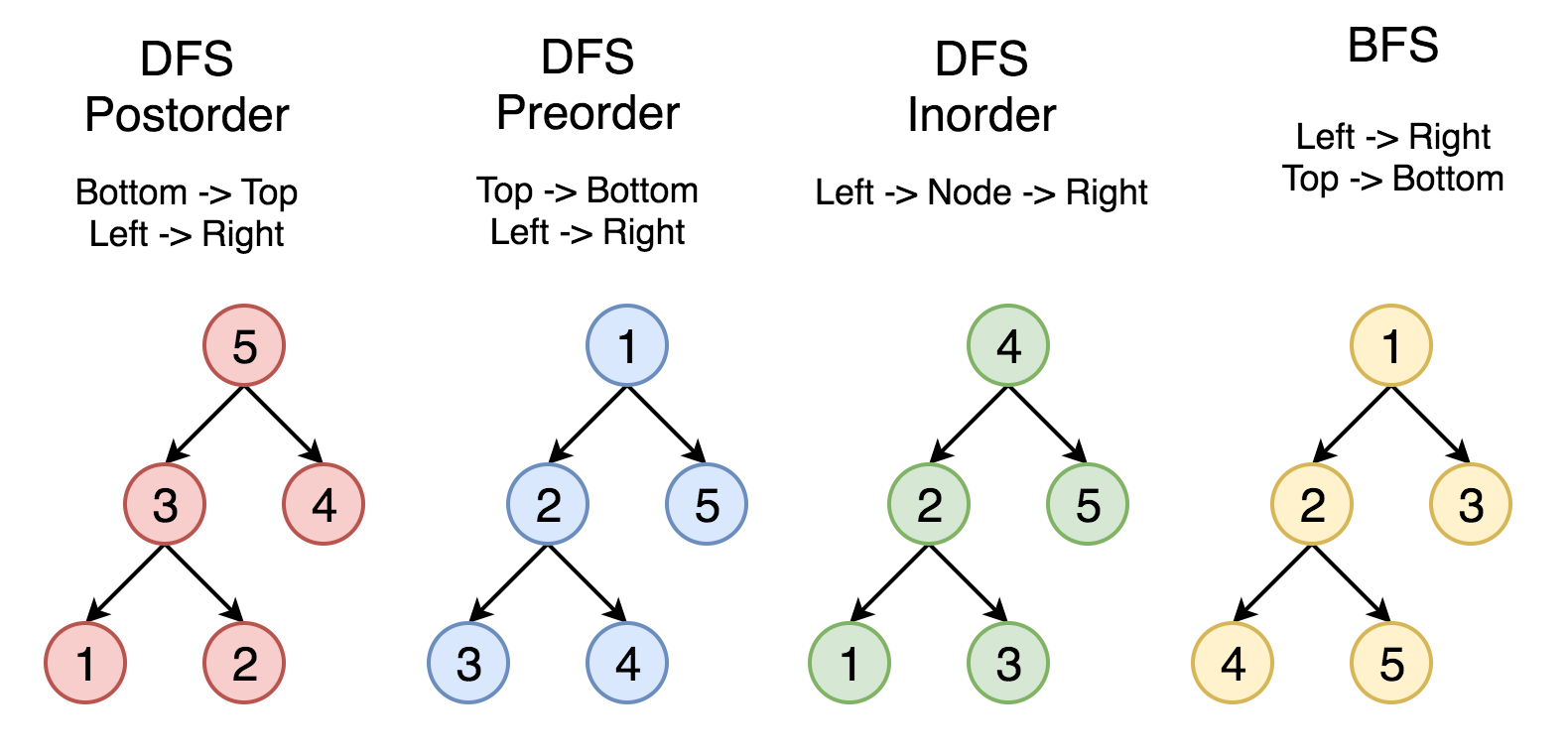
* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on a higher level would be visited before the ones on the lower levels.

* Depth First Search (DFS)

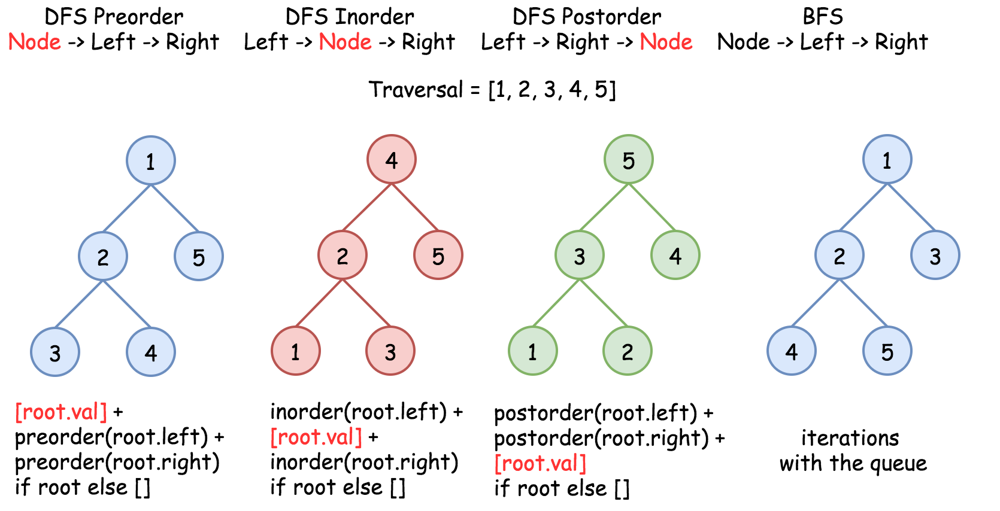
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach down to a leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node, and right node.

 Figure 1. The nodes are numerated in the order you visit them, please follow *1-2-3-4-5* to compare different strategies.

Here the problem is to implement postorder traversal using iterations.

#### **Approach 1: Recursive Postorder Traversal**

 Figure 2. Recursive DFS traversals.

The most straightforward way is to implement recursion using Left -> Right -> Node traversal strategy.

|  |
| --- |
| class Solution {  public void postorder(TreeNode root, List<Integer> output) {  if (root == null) return;  postorder(root.left, output);  postorder(root.right, output);  output.add(root.val);  }    public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> output = new ArrayList();  postorder(root, output);  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is the number of nodes. We visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*). Up to \mathcal{O}(H)O(*H*) to keep the recursion stack, where H*H* is a tree height. In the worst case of the skewed tree H = N*H*=*N*.

#### **Approach 2: Iterative Preorder Traversal: Tweak the Order of the Output**

Let's start from the root, and at each iteration, pop the current node out of the stack and push its child nodes. In the implemented strategy, we push nodes into stack following the order Top->Bottom and Left->Right. Since DFS postorder transversal is Bottom->Top and Left->Right the output list should be reverted after the end of the loop.

|  |
| --- |
| class Solution {  public List<Integer> postorderTraversal(TreeNode root) {  LinkedList<Integer> output = new LinkedList();  Deque<TreeNode> stack = new ArrayDeque();    if (root == null) return output;  stack.push(root);  while (!stack.isEmpty()) {  root = stack.pop();  output.addFirst(root.val);  if (root.left != null) stack.push(root.left);  if (root.right != null) stack.push(root.right);  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is the number of nodes. We visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*). Up to \mathcal{O}(H)O(*H*) to keep the stack, where H*H* is a tree height. In the worst case of the skewed tree H = N*H*=*N*.

#### **Approach 3: Iterative Postorder Traversal**

**Algorithm**

The idea is to fulfill the stack following right->node->left strategy. One could pop the last node out of the stack and check if it's the leftmost leaf. If yes, it's time to update the output. Otherwise, one should swap the last two nodes in the stack and repeat all these steps.

**Implementation**

|  |
| --- |
| class Solution {  public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> output = new ArrayList();  Deque<TreeNode> stack = new ArrayDeque();    while (root != null || !stack.isEmpty()) {  // push nodes: right -> node -> left  while (root != null) {  if (root.right != null) {  stack.push(root.right);  }  stack.push(root);  root = root.left;  }    root = stack.pop();    // if the right subtree is not yet processed  if (!stack.isEmpty() && root.right == stack.peek()) {  stack.pop();  stack.push(root);  root = root.right;  // if we're on the leftmost leaf  } else {  output.add(root.val);  root = null;  }  }  return output;  }  } |

**Complexity Analysis**

* Time complexity: \mathcal{O}(N)O(*N*), where N*N* is the number of nodes. We visit each node exactly once, thus the time complexity is \mathcal{O}(N)O(*N*).
* Space complexity: \mathcal{O}(N)O(*N*). Up to \mathcal{O}(H)O(*H*) to keep the stack, where H*H* is a tree height. In the worst case of the skewed tree H = N*H*=*N*.